

ND-Tree: a Fast Online Algorithm for Updating a Pareto Archive and its Application in Many-objective Pareto Local Search

Andrzej Jaszekiewicz^a, Thibaut Lust^b

^a*Poznan University of Technology, Faculty of Computing, Institute of Computing Science, ul. Piotrowo 2, 60-965 Poznan*

^b*Sorbonne Universités, UPMC, Université Paris 06, CNRS, LIP6, UMR 7606, F-75005, Paris, France*

Abstract

In this paper we propose a new method called ND-Tree for fast online update of a Pareto archive composed of mutually non-dominated solutions. ND-Tree uses a tree structure in which each node represents a subset of solutions contained in a hypercube defined by its local approximate ideal and nadir points. A leaf is a subset of solutions organized as a simple list, and an internal node is subset of solutions composed of the union of all its sub-nodes. Using heuristic rules we build subsets, either leaves or internal nodes, containing solutions located close in the objective space. Using basic properties of local ideal and nadir points we can efficiently avoid searching many branches in the tree. ND-Tree may be used in any multiobjective metaheuristics e.g. in an multiobjective evolutionary algorithm to update the external archive of potentially efficient solutions. We experimentally compare ND-Tree to simple list, quad-Tree, and M-Front methods using artificial and realistic benchmarks. Finally we apply ND-Tree within two-phase Pareto Local Search for traveling salesperson problems instances with up to 6 objectives. We show that with this new method substantial reduction of the computational time can be obtained.

Keywords: Multiobjective optimization, Pareto archive, Quad-tree, Pareto local search, Many-objective optimization

1. Introduction

In this paper we consider the problem of online update of a Pareto archive composed of mutually non-dominated solutions when a new candidate solution shows up. This problem is also referred to as dynamic non-dominance problem [27]. Online update of a Pareto archive is typically used in multiobjective metaheuristics (MOMHs), e.g. multiobjective evolutionary algorithms, whose goal is to generate a good approximation of the Pareto front. Most of the nowadays MOMHs use an external archive of potentially efficient solutions, i.e. Pareto archive containing solutions not dominated by any other solutions

Email addresses: `andrzej.jaszekiewicz@put.poznan.pl` (Andrzej Jaszekiewicz), `thibaut.lust@lip6.fr` (Thibaut Lust)

generated so far. We consider here MOMHs that generate iteratively new candidate solutions and use them to immediately update Pareto archive. Updating Pareto archive with a new solution x means that:

- all solutions dominated by x are removed from Pareto archive,
- x is added to Pareto archive if it was non-dominated w.r.t. to any solution in Pareto archive.

Online update of a Pareto archive may also be used as an intermediate step of other methods. For example Drozdík et al. [8] propose to use the online updated Pareto archive to speed-up non-dominated sorting procedure used in many multiobjective evolutionary algorithms.

The time needed to update Pareto archive generally grows with growing number of objectives and growing number of solutions. In some cases it may become a crucial part of the total running time of a MOMH.

The simplest data structure for storing Pareto archive is a plain list. When a new solution x is added, x is compared to all solutions in Pareto archive until either all solutions are checked or a solution dominating x was found.

In order to speed-up the process of updating a Pareto archive some authors proposed the use of specialized data structures and algorithms, e.g. quad-trees. However, the results of computational experiments reported in the literature are not conclusive and in some cases such data structures may in fact increase the update time compared to simple list.

The direct motivation for us to revisit the issue of updating Pareto archive comes from special properties of Pareto local search (PLS) method [25, 3]. PLS works directly with the Pareto archive. For each solution in the archive its neighborhood is searched to find new potentially efficient solutions and update the archive. Standalone PLS starting from random solutions is very inefficient since it spends a lot of time generating large numbers of solutions being still very far from the Pareto front. PLS, however, is used as a crucial component in some of the best methods for multiobjective knapsack [20], biobjective traveling salesperson problem (bTSP) [19, 18, 15] and set covering problem [21]. The general idea of such methods is to start PLS from a set of high quality solutions generated by some other methods, e.g. the powerful Lin-Kernighan heuristic for TSP [17].

PLS has also been used in other contexts, e.g. for solving scheduling problems [30, 9, 6], multi-agent problems [13] or multiobjective Markov decision processes [16].

The importance of PLS may be explained by the fact that unlike most other MOMHs it is especially good in the search for new solutions along Pareto front. In order to generate a good approximation of the Pareto front a MOMH has to achieve two different goals: the solutions generated by such method should both approach the Pareto front and cover all areas along this front. In other words, the MOMH has to search both towards and along Pareto front [1]. PLS seems to be especially good in search along Pareto front, i.e. when started from a sample of solutions located very close to the true Pareto front, PLS may generate a large number of new potentially efficient solutions in a very short time.

From the point of view of updating Pareto archive, PLS has some special characteristics:

- PLS generates large number of new candidate solutions in a very short time (as the neighborhood solutions for many problems may be quickly generated). In particular, before a neighbor solution is added to the archive only the new values of objectives are needed. These values may often be efficiently calculated using changes of the objectives. For example, in the case of TSP, the typical two edges exchange move requires only removing two edges and adding two other edges to the solution. So, the calculation of change of each objective requires just few arithmetic operations. This means that the time needed to generate a new solution may be much shorter than the time needed to update the archive.
- Since the new candidate solutions are neighbors of potentially efficient solutions they are usually quite good, i.e. even if the new solution is dominated there are only relatively few dominating solutions. In this case the plain list is especially inefficient since the first dominating solution will on average be found after checking large fraction of solutions.
- Pareto archive is a crucial element of PLS. New solutions are accepted or rejected based on the comparison to the archive. So Pareto archive has to be updated online, while in some other MOMHs it is at least potentially possible to store all candidate solutions and only at the end to filter out the dominated ones.

Based on these motivations, we propose a new method, called ND-Tree, for updating a Pareto archive. A thorough experimental study will show that we can obtain substantial computational time reductions comparing to state-of-the-art methods.

The remainder of the paper is organized as follows. Basic definitions related to multi-objective optimization are given in Section 2. In Section 3, we present a state of the art of the methods used for online updating a Pareto archive. The new method ND-Tree is described in Section 4. Finally, computational experiments are reported and discussed in Section 5.

2. Basic definitions

2.1. Multiobjective combinatorial optimization

We consider in this section a general multiobjective combinatorial optimization problem, defined as follows:

$$\begin{array}{ll}
\begin{array}{l} \text{“minimize”} \\ x \in \mathcal{X} \end{array} & y(x) = Cx \\
\text{subject to} & x : Ax \leq b \\
& x \in \{0, 1\}^n \\
x \in \{0, 1\}^n & \longrightarrow n \text{ variables,} \\
C \in \mathbb{N}^{p \times n} & \longrightarrow p \text{ objective functions} \\
A \in \mathbb{N}^{m \times n} \text{ and } b \in \mathbb{N}^{m \times 1} & \longrightarrow m \text{ constraints}
\end{array}$$

A combinatorial structure is associated to this problem, which can be path, tree, flow, tour, etc.

We denote by \mathcal{X} the feasible set in decision space, defined by $\mathcal{X} = \{x \in \{0, 1\}^n : Ax \leq b\}$. The image of the feasible set in objective space is called \mathcal{Y} and is defined by $\mathcal{Y} = y(\mathcal{X}) = \{Cx : x \in \mathcal{X}\} \subset \mathbb{N}^p$. Due to the contradictory features of the objectives, there does not exist a feasible solution simultaneously minimizing each objective (that is why the word minimize is placed between quotation marks) but a set of feasible solutions called *efficient solutions*. We present below some definitions that characterize these efficient solutions.

We first define dominance relations:

Definition 1. *Pareto dominance relation: we say that a vector $u = (u_1, \dots, u_p)$ dominates a vector $v = (v_1, \dots, v_p)$ if, and only if, $u_k \leq v_k \forall k \in \{1, \dots, p\} \wedge \exists k \in \{1, \dots, p\} : u_k < v_k$. We denote this relation by $u \prec v$.*

We can now define an efficient solution, a non-dominated point, the efficient set and the Pareto front.

Definition 2. *Efficient solution: a feasible solution $x^* \in \mathcal{X}$ is called efficient if there does not exist any other feasible solution $x \in \mathcal{X}$ such that $y(x) \prec y(x^*)$. For the sake of simplicity we will use the dominance relation w.r.t. solutions as well, i.e. $x \prec x^* \Leftrightarrow y(x) \prec y(x^*)$.*

Definition 3. *Non-dominated point: the image $y(x^*)$ in objective space of an efficient solution x^* is called a non-dominated point.*

Definition 4. *Efficient set: the efficient set denoted by \mathcal{X}_E contains all efficient solutions.*

Definition 5. *Pareto front: the image of the efficient set in \mathcal{Y} is called the Pareto front (or non-dominated frontier/set), and is denoted by \mathcal{Y}_N .*

Since in many places we will need dominance or equality relation we define it as coverage relation.

Definition 6. *Coverage relation: we say that a point u covers a point v if $u \prec v$ or $u = v$. We denote this relation by $u \preceq v$. We will also use the coverage relation w.r.t. solutions as well, i.e. $x \preceq x^* \Leftrightarrow y(x) \preceq y(x^*)$.*

Please note that coverage relation is sometimes referred to as weak dominance [5]. The terminology, however, is not consistent and other authors use weak dominance as the synonym of Pareto dominance [7] (in contrast to strong dominance where a solution is better on all objectives). Thus to avoid confusion we use the name “coverage”.

To define a Pareto archive, we need to introduce the mutually non-dominated relation:

Definition 7. *Mutually non-dominated relation: we say that two solutions are mutually non-dominated or non-dominated w.r.t. each other if none of the two solutions covers the other one.*

We can now define the Pareto archive:

Definition 8. *Pareto archive: set of solutions such that any pair of solutions of the set are mutually non-dominated.*

In the context of MOMHs, the Pareto archive contains thus the mutually non-dominated solutions generated so far (i.e. at a given iteration of a MOMH). We will denote $\hat{\mathcal{X}}_E$ the Pareto archive generated by a MOMH. In other words $\hat{\mathcal{X}}_E$ contains solutions that are potentially efficient at a given iteration.

The new method ND-Tree is based on *local* ideal and nadir points that we define below.

Definition 9. *Local ideal point of a subset $\mathcal{S} \in \mathcal{X}$ denoted as $z^*(\mathcal{S})$ is the point in the objective space composed of the best coordinates of all solutions belonging to \mathcal{S} , i.e. $z_k^*(\mathcal{S}) = \min_{x \in \mathcal{S}} \{y(x)_k\}, \forall k \in \{1, \dots, p\}$.*

Naturally, local ideal point covers all solutions in \mathcal{S} .

Definition 10. *Local nadir point of a subset $\mathcal{S} \in \mathcal{X}$ denoted as $z_*(\mathcal{S})$ is the point in the objective space composed of the worst coordinates of all solutions belonging to \mathcal{S} , i.e. $z_{*k}(\mathcal{S}) = \max_{x \in \mathcal{S}} \{y(x)_k\}, \forall k \in \{1, \dots, p\}$.*

Naturally, local nadir point is covered by all solutions in \mathcal{S} .

Approximations of these two points will be also used in ND-Tree:

Definition 11. *Approximate local ideal point of a subset $\mathcal{S} \in \mathcal{X}$ denoted as $\hat{z}^*(\mathcal{S})$ is a point in the objective space such that $\hat{z}^*(\mathcal{S}) \preceq z^*(\mathcal{S})$.*

Naturally, approximate local ideal point also covers all solutions in \mathcal{S} .

Definition 12. *Approximate local nadir point of a subset $\mathcal{S} \in \mathcal{X}$ denoted as $\hat{z}_*(\mathcal{S})$ is a point in the objective space such that $z_*(\mathcal{S}) \preceq \hat{z}_*(\mathcal{S})$.*

Naturally, approximate nadir ideal point is also covered by all solutions in \mathcal{S} .

3. State of the art

We present here a number of methods for the online update of Pareto archive presented in the literature and used in the comparative experiment. This review is not supposed to be exhaustive. Other methods can be found in [28] and reviews in [2, 23]. We describe two popular methods: linear list and quad-tree, the composite points method [10] and one recent method, M-Front [8], when used as a part of the non-dominated sorting procedure gave excellent performances compared to Jensen-Fortin's algorithm [14, 11], one of the fastest non-dominated sorting algorithm.

3.1. Linear List

3.1.1. General case

In this structure, a new solution is compared to all solutions of the list until a covering solution is found or all solutions are checked. The solution is only added if it is non-dominated w.r.t. all the solutions in the list, that is we need to browse whole list before adding a solution. The complexity in terms of number of Pareto dominance check is thus in $\mathcal{O}(N)$ with N the size of the list.

3.1.2. Biobjective case: sorted list

When only two objectives are considered, we can use the following special property: if we sort the list according to one objective (let's say the first), the non-dominated list is also sorted according to the second objective.

Therefore, roughly speaking, updating of the list can be efficiently done in the following way. Let us consider two minimized objectives. The list is kept sorted according to the first objective. We first determine the potential position of the new candidate solution in the sorted list according to its value for the first objective, with a binary search. Please note that in this case, it is more efficient to code the list with a dynamic array to get a constant access to the solutions of the list. Once the place of the new solution is found and it is not equal to some existing solution, it is enough to use these two procedures:

- We compare the new solution to the preceding one (if there is one) in the sorted list: this solution has a better value according to the first objective compared to the new solution. Therefore, if the preceding solution has also a better value according to the second objective, the new solution is dominated and cannot be added.
- If the solution is not dominated by the preceding solution, the solution can be added since the next solutions have a higher value for the first objective and cannot thus dominated the solution. We then need to check if there are some dominated solutions: we browse the next solutions of the list, until finding a solution that has a better evaluation according to the second objective compared to the new solution. All the solutions found that have a worse evaluation according to the second objective have to be removed since there are dominated by the new solution.

The worst-case complexity is still in $\mathcal{O}(N)$ since it can happen that a new solution has to be compared to all the other solutions (in the special case where we add a new solution in the first position and all the solutions of the sorted list are dominated by this new solution). But on average, experiments show that the behavior of this structure for handling two objectives updating problems is much better than a simple list since the worst-case scenario rarely occurs.

3.2. Quad-tree

The use of Quad-tree for storing potentially efficient solutions was proposed by Habenicht [12] and further developed by Sun and Steuer [29], and Mostaghim and Teich [22]. In Quad-tree, solutions are located in both internal nodes and leafs. Each node may have

p^2 sons corresponding to each possible combination of results of comparisons on each objective where a solution can either be better or not worse. In the case of mutually non-dominated solutions in fact $p^2 - 2$ sons are possible since the combinations corresponding to dominating or covered solutions are not possible. Quad-tree allows for a fast checking if a new solution is dominated or covered. A weak point of this data structure is that when an existing solution is removed its whole sub-tree has to be re-inserted to the structure. Thus, removal of a dominated solution is in general costly. In this paper we use Quad-tree2 version as described by Mostaghim and Teich [22].

3.3. M-Front

M-Front has been proposed relatively recently by Drozdík et al. [8] as a part of method for non-dominated sorting problem. The idea of M-Front is as follows. Assume that in addition to the new solution x a reference solution ref relatively close to x and belonging to the Pareto archive $\hat{\mathcal{X}}_E$ is known. The authors define two sets:

$$RS_U(x, ref) = \{y \in \hat{\mathcal{X}}_E \mid \exists k \in \{1, \dots, p\} : y_k \geq ref_k \wedge y_k \leq x_k\} \quad (1)$$

$$RS_L(x, ref) = \{y \in \hat{\mathcal{X}}_E \mid \exists k \in \{1, \dots, p\} : y_k \geq x_k \wedge y_k \leq ref_k\} \quad (2)$$

and prove that if a solution $y \in \hat{\mathcal{X}}_E$ is dominated by x then it belongs to $RS_L(x, ref)$ and if y dominates x then it belongs to $RS_U(x, ref)$. Thus, it is sufficient to compare the new solutions to sets $RS_L(x, ref)$ and $RS_U(x, ref)$ only. To find all solutions with objective values in a certain interval M-Front uses additional indexes one for each objective. Each index sorts the Pareto archive according to one objective.

To find a reference solution being close to x M-Front uses the k-d tree data structure. The k-d tree is a binary tree in which each intermediate node divides the space into two parts based on a value of one objective. While going down the tree the algorithm cycle over particular objectives, selecting one objective for each level. Drozdík et al. [8] suggest to store references to solutions in leaf nodes only, while intermediate nodes keep only split values. To insert a solution k-d tree selects either lower or upper-or-equal half-space.

Please note, that the paper of Drozdík et al. [8] does not give the precise algorithm of k-d tree. In our implementation when a leaf is reached a new division is made using the average value of the current level objective. The split value is average between the value for new solution and the solution in the leaf.

Also alike Drozdík et al. [8] approximate nearest neighbor is found exactly as in the standard exact nearest neighbor search, but only four evaluations of the distance are allowed.

We do not use the rebalancing of the k-d Tree described in [8] since it would never be activated in the case of all data sets used in this paper and is very unlikely to be used in the case of solutions generated by PLS.

3.4. M-Front-II

While implementing M-Front for the purpose of computational experiment we noticed that a number of elements can be improved to reduce further the running time. The improvement is in our opinion significant enough to call the new method M-Front-II (see Algorithm 1). The modifications we introduced are as follows:

- In original M-Front the sets $RS_L(x, ref)$ and $RS_U(x, ref)$ are built explicitly and only then the solutions contained in these sets are compared to x . In M-Front-II we build them only implicitly, i.e. we immediately compare the solutions that would be added to the sets to x .
- We analyze the objectives such that we start from objectives for which $ref_k \leq x_k$. In other words we start with solutions from set $RS_U(x, ref)$. Since many new solutions are dominated it allows to stop the search immediately when a solution dominating x was found. Note that a similar mechanism is in fact used in original M-Front but only after the sets $RS_L(x, ref)$ and $RS_U(x, ref)$ are explicitly built.
- The last modification is more technical. M-Front uses linked lists (`std::list` in C++) to store the indexes and a hash-table (`std::unordered_map` in C++) to link solutions with their positions in these lists. We observed, however, that even basic operations like iterating over the list are much slower with linked lists than with dynamic arrays (like `std::vector` in C++). Thus we use dynamic arrays for the indexes. In this case, however, there is no constant iterator that could be used to link the solutions with their positions in these indexes. So, we use the fast binary search to locate a position of a solution in the sorted index whenever it is necessary. The overhead of the binary search is anyway smaller than the savings due to use of faster hash-table.

3.5. Composite points

The composite points method has been developed by Fieldsend *et al* [10] in 2003. They introduce two new data structures called dominated tree and non-dominated tree. The dominated tree allows to detect if a new solution should be included into the Pareto archive (that is the new solution is not covered) and the non-dominated tree allows to determine which solutions of the Pareto archive are dominated by the new solution. Some particular points, called *composite points*, are used to build the trees. These points are based on the maximal and minimal coordinates of the points of the Pareto archive, and are built such that a total order between the composite points is obtained, according to the coverage relation. A binary search among the composite points of each tree allows then to update the Pareto archive when a new candidate point is added. This method has been compared to the simple list for standard multiobjective continuous problems (ZDT test functions [31]) but no significant difference in speed has been achieved.

Algorithm 1 M-Front-II Update

Parameter \uparrow : A Pareto archive $\hat{\mathcal{X}}_E$ with p indexes $I_k, k \in \{1, \dots, p\}$, i.e. lists containing references to all solutions sorted incrementally according to objective k and kd a k-d tree containing references to all solutions

Parameter \downarrow : New candidate solution x

if $\hat{\mathcal{X}}_E = \emptyset$ **then**

 Add x to $\hat{\mathcal{X}}_E$ and its reference to all indexes I_k and to kd

else

 Find reference solution ref using approximate nearest neighbor search in kd

if $ref \preceq x$ **then**

 STOP: x is rejected

else

 --| Comparison to set $RS_U(x, ref)$

for each $k \in \{1, \dots, p\} \mid ref_k < x_k$ **do**

 Find it , i.e. position of ref in index I_k using binary search

 Decrement it to the last position where $I_k[it]_k = ref_k$

while $I_k[it]_k \leq x_k$ **do**

if $I_k[it] \preceq x$ **then**

 STOP: x is dominated

$it++$

 Insert x to I_k right before it

 --| Comparison to set $RS_L(x, ref)$

for each $k \in \{1, \dots, p\} \mid ref_k > x_k$ **do**

 Find it , i.e. position of ref in index I_k using binary search

 Increment it to the last position where $I_k[it]_k = ref_k$

while $I_k[it]_k \geq x_k$ **do**

if $x \preceq I_k[it]$ **then**

 Remove $I_k[it]$ from $\hat{\mathcal{X}}_E$, indexes I_k and kd

$it--$

 Insert x to I_k right after it

if x was not covered by any solution in $\hat{\mathcal{X}}_E$ **then**

$\hat{\mathcal{X}}_E \leftarrow \hat{\mathcal{X}}_E + x$

 add x to kd

4. ND-tree

The new method proposed in this paper is based on the following idea that allows to reduce the number of comparisons to be made. Consider a subset $\mathcal{S} \subseteq \hat{\mathcal{X}}_E$ composed of mutually non-dominated solutions and a new candidate solution x . Assume that some approximate local ideal $\hat{z}^*(\mathcal{S})$ and approximate local nadir points $\hat{z}_*(\mathcal{S})$ are known. We can define the following simple properties:

Property 1. If x is covered by $\hat{z}_*(\mathcal{S})$, then x is covered by each solution in \mathcal{S} and thus can be rejected.

Proof 1. *This property is a straightforward consequence of the transitivity of the coverage relation.*

Property 2. If x covers $\hat{z}^*(\mathcal{S})$, then each solution in \mathcal{S} is covered by x .

Proof 2. *This property is also a straightforward consequence of the transitivity of the coverage relation.*

Property 3. If x is non-dominated w.r.t. both $\hat{z}_*(\mathcal{S})$ and $\hat{z}^*(\mathcal{S})$, then x is non-dominated w.r.t. each solution in \mathcal{S} .

Proof 3. *If x is non-dominated w.r.t. $\hat{z}_*(\mathcal{S})$ then there is at least one objective on which x is worse than $\hat{z}_*(\mathcal{S})$ and thus worse than each solution in \mathcal{S} . If x is non-dominated w.r.t. $\hat{z}^*(\mathcal{S})$ then there is at least one objective on which x is better than $\hat{z}^*(\mathcal{S})$ and thus better than each solution in \mathcal{S} . So, there is at least one objective on which x is better and at least one objective on which x is worse than each solution in \mathcal{S} .*

Property 4. If none of the above holds, i.e. x is neither covered by $\hat{z}_*(\mathcal{S})$, does not cover $\hat{z}^*(\mathcal{S})$, nor is non-dominated w.r.t. both $\hat{z}_*(\mathcal{S})$ and $\hat{z}^*(\mathcal{S})$, then all situations are possible, i.e. x may either be non-dominated w.r.t. all solutions in \mathcal{S} , covered by some solutions in \mathcal{S} or dominate some solutions in \mathcal{S} .

Proof 4. *This property can be proven by showing examples of each of the situation. Consider for example a set $\mathcal{S} = \{(1, 1, 1), (0, 2, 2), (2, 2, 0)\}$ with $\hat{z}^*(\mathcal{S}) = z^*(\mathcal{S}) = (0, 1, 0)$ and $\hat{z}_*(\mathcal{S}) = z_*(\mathcal{S}) = (2, 2, 2)$. A new solution $(1, 1, 0)$ dominates a solution in \mathcal{S} , a new solution $(1, 1, 2)$ is dominated (thus covered) by a solution in \mathcal{S} , and solutions $(0, 3, 0)$ and $(2, 0, 1)$ are non-dominated w.r.t. all solutions in \mathcal{S} .*

The properties are graphically illustrated for the biobjective case in Figure 1. As can be seen in this figure, in the biobjective case, if x is covered by $\hat{z}^*(\mathcal{S})$ and x is non-dominated w.r.t. $\hat{z}_*(\mathcal{S})$ then x is dominated by at least one solution in \mathcal{S} . Note, however, that this does not hold in the case of three and more objectives as shown in the example used in the proof - the point $(0, 3, 0)$ is covered by $\hat{z}^*(\mathcal{S}) = (0, 1, 0)$, non-dominated w.r.t. $\hat{z}_*(\mathcal{S}) = (2, 2, 2)$ and $(0, 3, 0)$ is not dominated by any points in \mathcal{S} .

In fact it is possible to distinguish more specific situations if property 4 holds, e.g. situation when a new solution may be covered but cannot dominate any solution, but since we do not distinguish them in the proposed algorithm we do not define them formally.

The above properties allow in some cases to quickly compare a new candidate solution x to all solutions in a set \mathcal{S} without the need for further comparisons to individual solutions

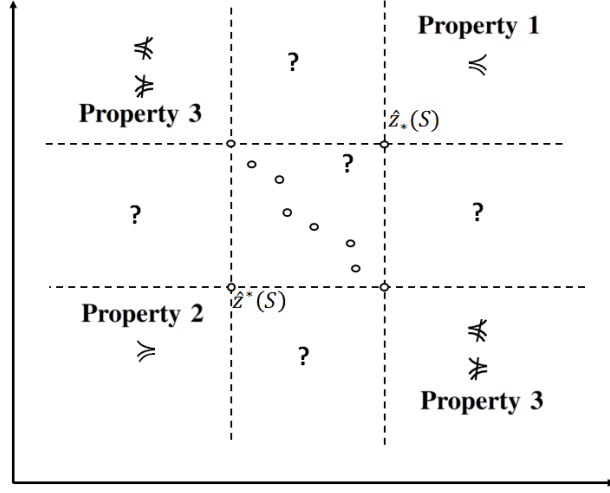


Figure 1: Comparison of a new solution to all solutions in set \mathcal{S} based on comparisons to $\hat{z}^*(\mathcal{S})$ and $\hat{z}_*(\mathcal{S})$ only.

belonging to \mathcal{S} . Such further comparisons are necessary only if the situations described by Property 4 hold. Intuitively, the closer the approximate local ideal and nadir points the stronger are these properties, i.e. it is more likely that the further comparisons can be avoided. To obtain close approximate local ideal and nadir points we should:

- Split the whole set of potentially efficient solutions into subsets of solutions located close in the objective space.
- Have good approximations of the exact local ideal and nadir points. On the other hand calculation of the exact points may be computationally demanding and a reasonable approximation may assure the best overall efficiency.

Based on these properties, we can now define the ND-Tree structure.

Definition 13. *ND-Tree data structure is a tree with the following properties:*

1. *With each node n is associated a set of solutions $\mathcal{S}(n)$.*
2. *Each leaf node contains a list $\mathcal{L}(n)$ of solutions and $\mathcal{S}(n) = \mathcal{L}(n)$.*
3. *For each internal node n , $\mathcal{S}(n)$ is the union of sets associated with all sons of n .*
4. *Each node n stores an approximate ideal point $\hat{z}^*(\mathcal{S}(n))$ and approximate nadir point $\hat{z}_*(\mathcal{S}(n))$.*
5. *If n' is a son of n , then $\hat{z}^*(\mathcal{S}(n)) \preceq \hat{z}^*(\mathcal{S}(n'))$ and $\hat{z}_*(\mathcal{S}(n')) \preceq \hat{z}_*(\mathcal{S}(n))$.*

The algorithm to update a Pareto archive with ND-Tree is given in Algorithm 2 (**Update**). The idea of the algorithm is as follows. We start by checking if the new solution x is dominated, covered or non-dominated w.r.t. all solutions in $\hat{\mathcal{X}}_E$ by going through the nodes of ND-Tree and skipping sons (and thus their sub-trees) for which property 4 does not hold. This procedure is presented in Algorithm 3 (**UpdateNode**).

The new solution is first compared to the approximate ideal point ($\hat{z}^*(\mathcal{S}(n))$) and nadir point ($\hat{z}_*(\mathcal{S}(n))$) of the current node. If the new solution is dominated by $\hat{z}_*(\mathcal{S}(n))$ it is immediately rejected (**Property 1**). If $\hat{z}^*(\mathcal{S}(n))$ is covered, the node is deleted and its whole sub-tree as well (**Property 2**). Otherwise if $\hat{z}^*(\mathcal{S}(n)) \preceq x$ or $x \preceq \hat{z}_*(\mathcal{S}(n))$ (**Property 4**) and if n is a leaf node, x may be dominated by or dominate some solutions of n and it is necessary to browse the whole list $\mathcal{L}(n)$ of the node n . If a solution dominating x is found, x is rejected, and if a solution dominated by x is found, the solution is deleted from $\mathcal{L}(n)$.

If after checking ND-Tree the new solution was found to be non-dominated it is inserted by adding it to a close leaf (Algorithm 4 **Insert**). To find a proper leaf we start from the root and always select a son with closest distance to x . As a distance measure we use the Euclidean distance to the middle point between approximate ideal and approximate nadir points. The middle point is a point with individual coordinates equal to the average of corresponding coordinates in approximate ideal and approximate nadir points. Such distance measure is very fast to compute based on the two points only. Since we do not need the value of the Euclidean distance but just the order induced by this distance, we used squared Euclidean distance to avoid calculation of the root. Once we have reached a leaf node, we add the solution x to the list $\mathcal{L}(n)$ of the node and possibly update the ideal and nadir points of the node n (Algorithm 6 **UpdateIdealNadir**). However, if the size of $\mathcal{L}(n)$ became larger than the maximum allowed size of a leaf set, we need to split the node into a predefined number of sons (Algorithm 5 **Split**). We first select one solution of $\mathcal{L}(n)$ for each new subnode. The first solution selected is the one with the highest Euclidean distance to all other solutions in $\mathcal{L}(n)$. Then we select the solution with the highest average Euclidean distance to all the solutions contained in all subnodes already created. We continue this procedure until the required number of subnodes is created. Finally, we add each remaining solution of $\mathcal{L}(n)$ to the closest subnode. We use as distance measure the Euclidean distance to the middle point between approximate ideal and approximate nadir points, as done in the **Insert** algorithm.

The approximate local ideal and nadir points are updated only when a solution is added. We do not update them when solution(s) are removed since it is a more complex operation. This is why we deal with approximate (not exact) local ideal and nadir points.

Please note that our goal is to propose a practically fast method. We do not present any analysis of worst case complexity but we will show that this method performs well in computational experiments.

Algorithm 2 Update

Parameter \uparrow : A Pareto archive $\hat{\mathcal{X}}_E$ organized as ND-Tree

Parameter \downarrow : New candidate solution x

if $\hat{\mathcal{X}}_E = \emptyset$ **then**

 Create a leaf node n with an empty list set $\mathcal{L}(n)$ and use it as a root

$\mathcal{L}(n) \leftarrow \mathcal{L}(n) + x$

else

$n \leftarrow$ root node

 UpdateNode($n \uparrow, x \downarrow$)

if x was not covered by any solution in $\hat{\mathcal{X}}_E$ **then**

 Insert($n \uparrow, x \downarrow$)

Algorithm 3 UpdateNode

Parameter \uparrow : A node n

Parameter \downarrow : New candidate solution x

Compare x to $\hat{z}^*(\mathcal{S}(n))$ and $\hat{z}_*(\mathcal{S}(n))$

if $\hat{z}_*(\mathcal{S}(n)) \preceq x$ **then**

 --| Property 1

 STOP: x is rejected

else if $x \preceq \hat{z}^*(\mathcal{S}(n))$ **then**

 --| Property 2

 Remove n and its whole sub-tree

else if $\hat{z}^*(\mathcal{S}(n)) \preceq x$ **or** $x \preceq \hat{z}_*(\mathcal{S}(n))$ **then**

 --| Property 4

if n is a leaf node **then**

for each $y \in \mathcal{L}(n)$ **do**

if $y \preceq x$ **then**

 STOP: x is rejected

else if $x \prec y$ **then**

 Remove y

else

for each Subnode n' of n **do**

 UpdateNode ($n' \uparrow, x \downarrow$)

if n' became empty **then**

 Remove n'

5. Computational experiments

We will show results obtained with ND-Tree and other methods in three different cases:

Algorithm 4 Insert

Parameter \uparrow : A node n

Parameter \downarrow : New candidate solution x

if n is a leaf node **then**

$\mathcal{L}(n) \leftarrow \mathcal{L}(n) + x$

 UpdateIdealNadir ($n \uparrow, x \downarrow$)

if Size of $\mathcal{L}(n)$ became larger than maximum size of a leaf set **then**

 Split ($n \uparrow$)

else

 Find subnode n' of n being closest to x

 Insert($n' \uparrow, x \downarrow$)

Algorithm 5 Split

Parameter \uparrow : A node n

Find the solution $y \in \mathcal{L}(n)$ with the highest average Euclidean distance to all other solutions in $\mathcal{L}(n)$

Create a new subnode n' with an empty list set $\mathcal{L}(n')$

$\mathcal{L}(n') \leftarrow \mathcal{L}(n') + y$

UpdateIdealNadir ($n' \uparrow, y \downarrow$)

$\mathcal{L}(n) \leftarrow \mathcal{L}(n) - y$

while The required number of subnodes are not created **do**

 Find the solution $y \in \mathcal{L}(n)$ with the highest average Euclidean distance to all solutions in all subnodes of n

 Create a new subnode n' with an empty list set $\mathcal{L}(n')$

$\mathcal{L}(n') \leftarrow \mathcal{L}(n') + y$

 UpdateIdealNadir ($n' \uparrow, y \downarrow$)

$\mathcal{L}(n) \leftarrow \mathcal{L}(n) - y$

while $\mathcal{L}(n)$ is not empty **do**

$y \leftarrow$ first solution in $\mathcal{L}(n)$

 Find subnode n' of n being closest to y

$\mathcal{L}(n') \leftarrow \mathcal{L}(n') + y$

 UpdateIdealNadir ($n' \uparrow, y \downarrow$)

$\mathcal{L}(n) \leftarrow \mathcal{L}(n) - y$

- A) Results for sets artificially generated, which allow us to easily control the number of solutions of the sets and the quality of the solutions.
- B) Results when ND-Tree is integrated into PLS for solving the multiobjective traveling salesperson problem.

Algorithm 6 UpdateIdealNadir

Parameter \uparrow : A node n

Parameter \downarrow : New candidate solution x

Check in any component of x is lower than corresponding component in $\hat{z}^*(\mathcal{S}(n))$ or greater than corresponding component in $\hat{z}_*(\mathcal{S}(n))$ and update the points if necessary
if $\hat{z}^*(\mathcal{S}(n))$ or $\hat{z}_*(\mathcal{S}(n))$ have been changed **then**

if n is not a root **then**

$np \leftarrow \text{parent of } n$

 UpdateIdealNadir ($np \uparrow, x \downarrow$)

C) Results for sets coming from solutions generated by PLS for solving the multiobjective traveling salesperson problem.

To avoid the influence of implementation details all methods were implemented from the scratch in C++ and C in as much homogeneous way as possible, i.e. when possible the same code was used to perform the same operations like Pareto dominance checks. The code, as well as test instances and data sets, are available from the authors upon request. Most of the results have been obtained on an Intel Core i7-5500U CPU at 2.4 GHz. The results involving PLS (Section 5.2) were obtained on an i5-450M CPU at 2.4 GHz.

5.1. Artificial sets

The artificial sets are composed of n points with p objectives to minimize. The sets are created as follows. We generate randomly n points y^i in $\{0, \dots, V_{max}\}^p$ with the following constraint (otherwise there can be few non-dominated points, all located near to the point $(0, \dots, 0)$): $\sum_{k=1}^p (V_{max} - y_k^i)^2 \leq V_{max}^2$. With these constraints, all the non-dominated points will be close to the hypersphere of center $(V_{max}, \dots, V_{max})$ and with a radius of length equal to V_{max} . In order to control the quality of the points generated, we also add a quality constraint: $\sum_{i=k}^p (V_{max} - y_k^i)^2 \geq (1 - \epsilon) * V_{max}^2$. With a small ϵ , only high-quality solutions will be generated and this simulates well the behavior of a MOMH.

We have generated data sets composed of 100 000 and 200 000 points, with $V_{max} = 10000$, and for $p = 2$ to 6. In the main experiment we use data sets with 100 000 points because for the larger sets running times of some methods became very long.

For each value of p , five different quality sets are considered:

- Quality = 1, $\epsilon = 0.5$
- Quality = 2, $\epsilon = 0.25$
- Quality = 3, $\epsilon = 0.1$
- Quality = 4, $\epsilon = 0.05$

Table 1: Numbers of non-dominated solutions in artificial sets

p	Quality	number of non-dominated solutions
2	1	519
2	2	713
2	3	1046
2	4	1400
2	5	2735
3	1	4588
3	2	6894
3	3	12230
3	4	19095
3	5	53813
4	1	14360
4	2	21680
4	3	39952
4	4	64664
4	5	98283
5	1	28944
5	2	42246
5	3	77477
5	4	96002
5	5	99975
6	1	45879
6	2	65195
6	3	96687
6	4	99788
6	5	100000

- Quality = 5, $\epsilon = 0.01$

The fraction of non-dominated solutions grows both with increasing quality and number of objectives and in extreme cases all solutions may be non-dominated (see table 1).

We compare simple list, sorted list (biobjective case), Quad-tree, M-Front, M-Front-II and ND-Tree for these sets according to the CPU time (ms).

For ND-Tree we use 20 as the maximum size of leaf and $p + 1$ as the number of sons. These values of the parameters were found to perform well in many cases. We analyze the influence of these parameters later in this section.

Each method was run 10 times for each set each time processing the solutions in a different random order. The average running times are presented in Figures 2 to 6. Please note that because of large differences the running time is presented in logarithmic scale. In addition, in Figure 7 we illustrate the evolution of the running times according to the

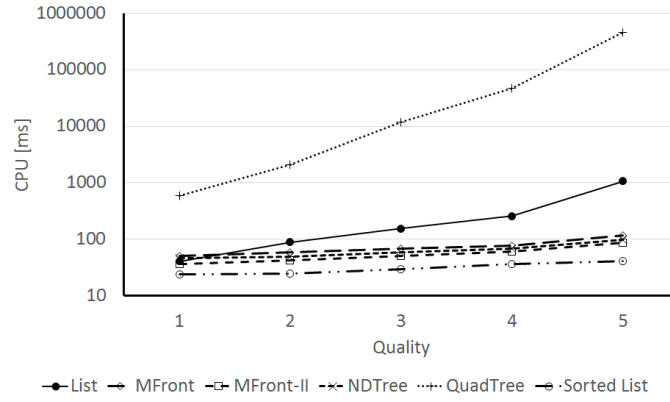


Figure 2: CPU time (logarithmic scale) for biobjective data sets

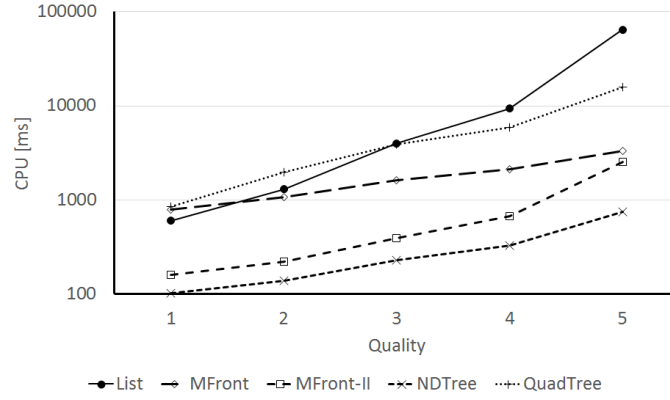


Figure 3: CPU time (logarithmic scale) for three-objective data sets

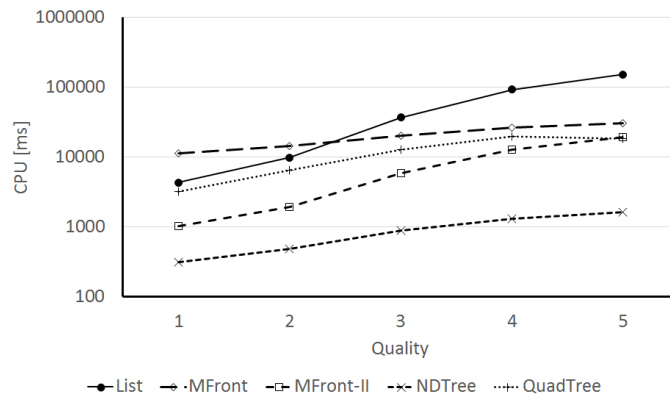


Figure 4: CPU time (logarithmic scale) for four-objective data sets

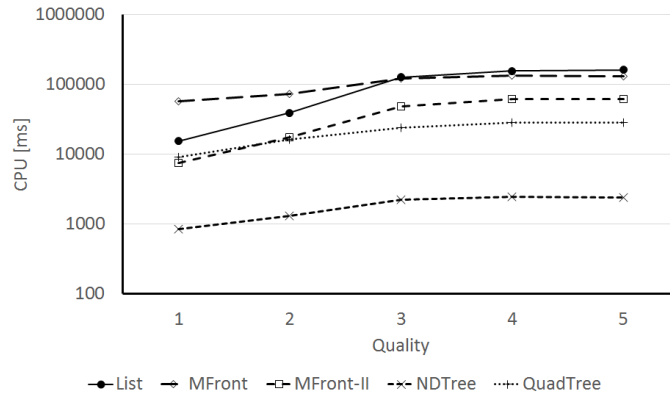


Figure 5: CPU time (logarithmic scale) for five-objective data sets

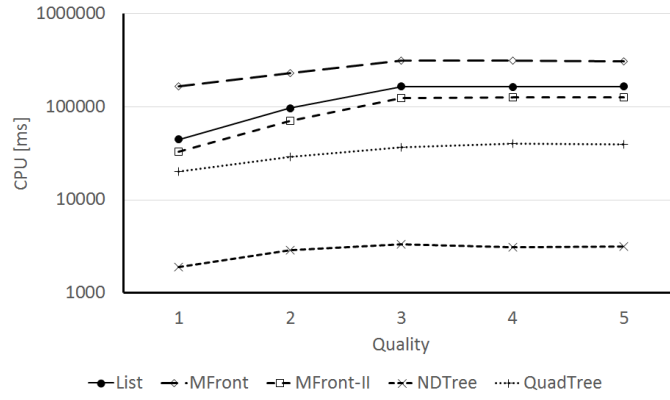


Figure 6: CPU time (logarithmic scale) for six-objective data sets

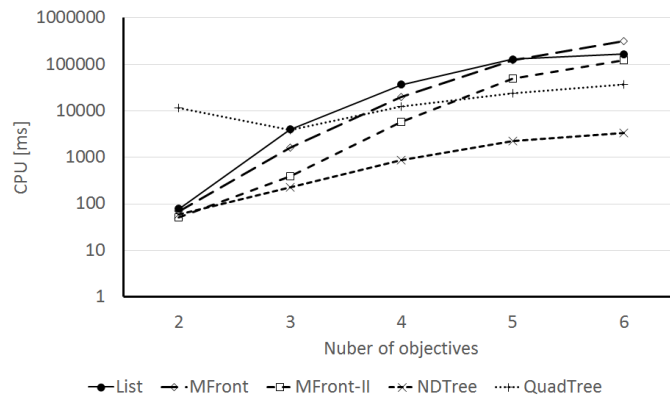


Figure 7: CPU time (logarithmic scale) vs. number of objectives for data sets of quality 3

number of objectives for the data sets of intermediate quality 3.

The main observations from this experiment are:

- ND-Tree performs the best for all test sets with three and more objectives. In some cases the differences to other methods are of two orders of magnitude and in some cases the difference to the second best method is of one order of magnitude. ND-Tree behaves also very predictably, its running time grows slowly with increasing number of objectives and increasing fraction of non-dominated solutions.
- For biobjective instances sorted list is the best choice. In this case, M-Front and M-Front-II also behave very well since they become very similar to sorted list.
- Simple list obtains its best performances for data sets with many dominated solutions like $p = 2$ with lowest quality. In this case the new solution is dominated by many solutions, so the search process is quickly stopped after finding a dominating solution.
- Quad-tree performs very bad for data sets with many dominated solutions, e.g. on biobjective instances where it is worst in all cases. In this case, many solutions added to Quad-tree are then removed and the removal of a solution from Quad-tree is a costly operation as discussed above. On the other hand, it is the second best method for all data sets with six objectives.
- M-Front-II is much faster than M-Front on data sets with larger fraction of dominated solutions. In this case, M-Front-II may find a dominating solution faster without building explicitly the whole sets $RS_L(x, ref)$ and $RS_U(x, ref)$.
- The performance of both M-Front and M-Front-II deteriorates with increasing number of objectives. With six objectives M-Front is the slowest method in all cases. Intuitively this can be explained by the fact that M-Front (both versions) uses each objective individually to reduce the search space. In case of two objectives the values of one objective carry a lot of information since order on one objective induces also an order on the other one. The more objectives, the less information we get from an order on one of them. Furthermore, sets $RS_L(x, ref)$ and $RS_U(x, ref)$ are in fact unions of corresponding sets for particular objectives which also results in their growth. Finally, M-Front is based on the assumption that a reference solution close on Euclidean distance is also close on each objective. The more objectives, the less it is so, since the close solution will rather have a good balance of differences on many coordinates, but it does not have to be particularly close on individual objectives.

In an additional experiment we analyzed also the evolution of the running time of all methods with increasing number of solutions. We decided to use one intermediate data set with $p = 4$ and quality 3. We used 200 000 solutions in this case and 10 runs for each method. The results are presented in Figure 8. In addition, since the running time

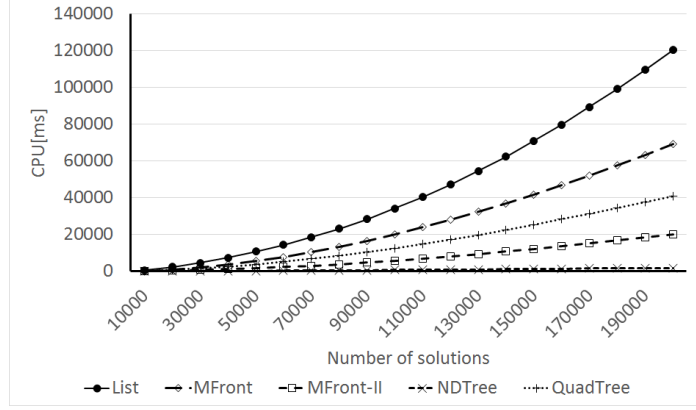


Figure 8: CPU time vs. the number of solutions for four-objective data sets of quality 3

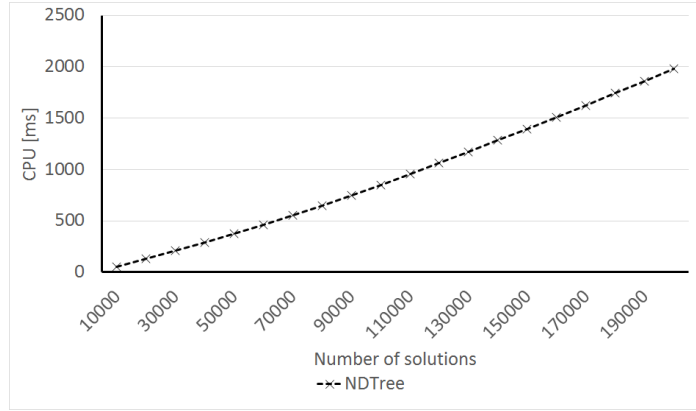


Figure 9: CPU time vs. the number of solutions for ND-Tree only (four-objective data sets of quality 3)

is much smaller for ND-Tree its results are presented in Figure 9 separately. We see that ND-Tree is the fastest method for any number of solutions and its running time grows almost linearly with the number of solutions.

ND-Tree has two parameters - the maximum size (number of solutions) of a leaf, and the number of sons, so the question arises how sensitive it is to the setting of these parameters. To study it we again use the intermediate data set with $p = 4$ and quality 3 and run ND-Tree with various parameters. The results are presented in Figure 10. Please note, that number of sons cannot be larger to maximum size of leaf +1 since after exceeding the maximum size the leaf is split into the given number of sons. We see that ND-Tree is not very sensitive to the two parameters: the CPU time remains between about one and three seconds regardless the values of the parameters. The best results are however obtained with 20 for the maximum size of the list and with 6 for the number of sons.

In our opinion the results confirm that ND-Tree performs relatively well for a wide range of the values of the parameters. In fact, it would remain the best method for this data set with any of the parameters settings tested.

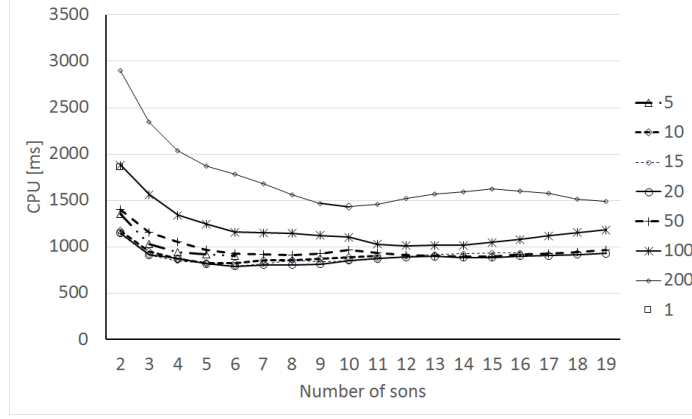


Figure 10: ND-Tree CPU time for different values of parameters (four-objective data sets of quality 3). Data series correspond to maximum leaf size.

5.2. Pareto local search

In this section, we solve instances of the multiobjective traveling salesperson problem (MOTSP) with PLS. We use this problem since state-of-the-art methods for biobjective TSP use PLS as one of the components (see [19, 15]). We compare the results of ND-Tree with the simple (sorted) list data structure (commonly used in PLS). We do not use other methods since we use an existing efficient implementation of PLS for MOTSP in C¹, and full integration of all methods with this implementation would be technically very complex. However, in the next section we will compare all methods on sets of points generated by PLS.

There are different versions of PLS according to the solutions from which the neighborhood is applied. There are versions where the neighborhood is only applied from non-dominated solutions [25] and there are versions where it is allowed to apply the neighborhood from dominated solutions [3, 13].

For experimental reasons explained latter, we will use a version where the neighborhood can be applied from dominated solutions.

Our version of PLS works as follows. The method starts with a population P composed of potentially efficient solutions given by an initial population P_0 . Then, all the neighbors p' of each solution p of P are generated. If a neighbor p' is not covered by the current solution p , we update a local list L_n of non-dominated neighbors with p' . Once all the neighbors p' of p have been generated, we update \mathcal{X}_E with L_n . We found it more efficient to use a local list of non-dominated neighbors rather than updating \mathcal{X}_E each time when a new neighbor is produced (as commonly done in PLS, see [25, 3, 19]). The neighbors list L_n is reinitialized after each neighborhood exploration. Finally, we save the neighbors that have been added to \mathcal{X}_E , to an auxiliary population P_a . We then start again the process with $P = P_a$ until P is empty. The pseudo-code of PLS is given by Algorithm 7.

¹Available at <http://www-desir.lip6.fr/~lustt>

In the algorithm, $\mathcal{N}(p)$ denotes the neighborhood of p and **Update()** is the procedure used to update Pareto archive with a new solution (and returns true if the new solution has been added to this set).

Please note that the **Update()** procedure is used in two different contexts: first **Update()** is used to update the local list of non-dominated neighbors and secondly to update the Pareto archive. For the first case, we have noticed in our experiments that the size of the neighbors list was quite small (less than 50). Therefore, in all cases, a simple linear list has been used to store the non-dominated neighbors. Also, in some versions of PLS [3, 19, 18], the auxiliary population P_a is also updated to keep only the non-dominated solutions. In the version of PLS used in this paper, we simply add a solution to P_a without any dominance checking. The main reason is that, in this way, the only difference between the use of two different data structures in PLS comes from the update of the Pareto archive.

Algorithm 7 PLS

Parameter \downarrow : An initial population P_0 of non-dominated solutions
Parameter \uparrow : A Pareto archive $\hat{\mathcal{X}}_E$

$P \leftarrow P_0$
 $\hat{\mathcal{X}}_E \leftarrow P$
--| Initialization of an auxiliary population P_a
 $P_a \leftarrow \emptyset$
while $P \neq \emptyset$ **do**
--| Generation of all the neighbors p' of all the solutions $p \in P$
for each $p \in P$ **do**
--| Initialization of a list of neighbors L_n
 $L_n \leftarrow \emptyset$
for each $p' \in \mathcal{N}(p)$ **do**
if $y(p) \not\leq y(p')$ **then**
 $\text{Update}(L_n \uparrow, p' \downarrow)$
for each $n \in L_n$ **do**
if $\text{Update}(\hat{\mathcal{X}}_E \uparrow, n \downarrow)$ **then**
 $P_a \leftarrow P_a + n$
--| P is composed of the solutions of P_a
 $P \leftarrow P_a$
--| Re-initialization of P_a
 $P_a \leftarrow \emptyset$

We have used Euclidean MOTSP instances with $p = 2$ to $p = 6$ objectives. The size of the instances are given in Table 2. The p costs between the edges correspond to the Euclidean distance between two points in a plane.

We have considerably reduced the size of the instances according to the number of

Table 2: MOTSP instances

p	2	3	4	5	6
Size	500	50	25	20	15

Table 3: Size of the initial population

p	2	3	4	5	6
$ NDP $	836	411	428	489	263

objectives. Indeed, when the number of objectives increase, the number of Pareto non-dominated generally increases also significantly.

It is now well-known that PLS needs a good-quality initial population to start [19]. Otherwise, PLS converges very slowly to the Pareto front. We have used as initial population for all the instances, the potentially efficient solutions obtained from 1000 Lin-Kernighan runs [4] with a linear aggregation of the objectives and random weight sets. The number of potentially efficient solutions obtained ($|NDP|$) in this way is given in Table 3.

We have used for the neighborhood simple 2-opt moves (two edges are removed, and two new edges are added). However, as generating all the 2-opt moves for each solution of the population will be very time-consuming ($\frac{N(N-3)}{2}$ neighbors to generate for each solution, with N cities), we only consider a small proportion of all the possible 2-opt moves. As done in [18], candidate lists for the 2-opt moves are used: the candidate lists are created on the basis of the edges used by the solutions of the initial population. More precisely, we explore the set of candidate edges of the initial population, and for each candidate edge $\{v_i, v_j\}$, we add the city v_j to the candidate list of the city v_i . Moreover, for the instances with at least four objectives, we limit the size of the candidate list to 2 to limit the number of neighbors generated in order to keep reasonable running times (which can be very high when the list is used to update the Pareto archive).

We also introduce another speed-up technique in PLS for solving the MOTSP: in the neighbors list L_n , we only save the new objective values and the indexes, in the current solution, of the two edges to be removed, but we do not modify the whole representation of the solution. Then the neighbor solution is fully built only if it is non-dominated according to the Pareto archive $\hat{\mathcal{X}}_E$. That enables to reduce the computational time of PLS since building a new solution from a 2-opt move is in $\mathcal{O}(N)$ and we avoid this operation for new solutions that are found to be covered.

The result of the comparison between the use of the linear list and ND-Tree for solving the MOTSP with PLS is given in Table 4.

The results illustrate the substantial benefit of using ND-Tree in comparison to the simple list: by only changing the method used to update the Pareto archive we obtain

Table 4: Comparison between the list and ND-Tree for updating the potentially efficient solutions set in PLS. *In the biobjective case, the sorted list structure has been used.

Instance			CPU(s)		Speed-up
p	Size	$ NDP $	List	ND-Tree	
2	500	32842	24.26*	24.82	0.98
3	50	94343	3517.52	33.26	106
4	25	84013	1299.30	18.81	69
5	20	359581	20130.35	282.14	71
6	15	49341	257.93	12.09	21

speed-up factors that goes to 21 (6 objectives instance) to 106 (3 objectives instance). For example, for the 5 objectives instance, PLS with the list needs more than 5 hours while PLS with ND-Tree only needs not more than 5 minutes. On the other hand, for the two objectives instance, we have used the sorted list, and in this case, ND-Tree does not allow to improve the running time of PLS.

5.3. Sets generated by PLS

In this section we compare ND-Tree with the list, quad-tree, M-Front and M-Front-II for sets of points generated by PLS. The points correspond to:

- the initial points obtained from 1000 Lin-Kernighan runs,
- points generated with the application of the neighborhood from the initial points (i.e. one phase of PLS).

The numbers of all solutions and non-dominated solutions are given in Table 5.

The results are presented in Figure 11. Please note that the results are not directly comparable between different numbers of objectives since the sets differ in total number of solutions. The results confirm that the observations made for artificial sets also hold in the case of real sets. ND-Tree is the fastest method for three and more objectives. Quad-tree performs particularly bad in biobjective case. Both versions of M-Front relatively deteriorate with growing number of objectives.

6. Conclusion

According to the results of the computational experiments ND-Tree should be a method of choice for storing and updating a Pareto archive in the case of three and more objectives problems. In biobjective case the best choice remains the sorted list.

We believe that with the proposed method for updating a Pareto archive, new state-of-the art results could be obtained for many MOCO problems (with $p > 2$) and some MOMHs could be adapted to this special data structure in order to improve their results. In fact in this paper we were able to apply PLS to MOTSP instances with up to 6

Table 5: Numbers of solutions in sets generated by PLS

p	N	Number of solutions	Number of non-dominated solutions
2	500	593579	6471
3	100	199149	22960
4	50	133981	37561
5	35	105901	47691
6	25	45223	20663

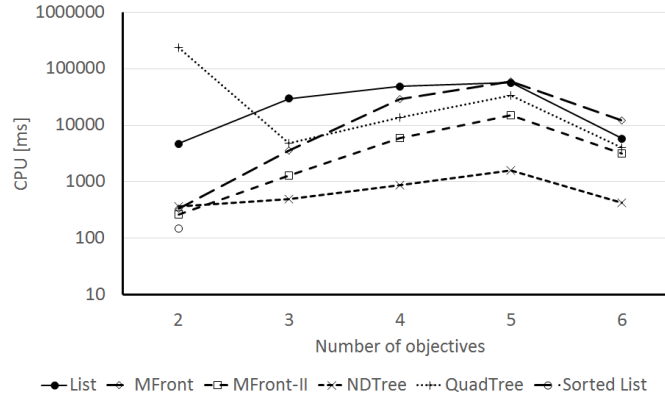


Figure 11: CPU time (logarithmic scale) on sets generated by PLS

objectives while according to our knowledge so far PLS was applied to biobjective and three-objective instances only.

An interesting direction for further research is to adapt ND-Tree to be able to deal with archives of a relatively large but limited size. Finally, it would be interesting to study how ND-tree could be modified to update archives with stronger dominance relations than the Pareto dominance (e.g. sorted-Pareto dominance [24] or Lorenz dominance [26]). In this way, preferences of a decision maker could be integrated and methods such as PLS will not be limited by the high number of Pareto efficient solutions when solving many-objective combinatorial optimization problems.

Acknowledgment

The research of Andrzej Jaszkievicz was funded by the the Polish National Science Center, grant no. UMO-2013/11/B/ST6/01075.

References

- [1] A.Lara, Sanchez, G., Coello, C. C., Schütze, O., Feb 2010. HCS: A new local search strategy for memetic multiobjective evolutionary algorithms. IEEE Transactions on Evolutionary Computation 14 (1), 112–132.

- [2] Altwaijry, N., Bachir Menai, M., 2012. Data structures in multi-objective evolutionary algorithms. *Journal of Computer Science and Technology* 27 (6), 1197–1210.
- [3] Angel, E., Bampis, E., Gourvès, L., 2004. A dynasearch neighborhood for the bi-criteria traveling salesman problem. In: Gandibleux, X., Sevaux, M., Sörensen, K., T’kindt, V. (Eds.), *Metaheuristics for Multiobjective Optimisation*. Springer. Lecture Notes in Economics and Mathematical Systems Vol. 535, Berlin, pp. 153–176.
- [4] Applegate, D., 2003. Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing* 15, 82–92.
- [5] Brockhoff, W., 2010. Theory of Randomized Search Heuristics: Foundations and Recent Developments. World Scientific Publishing Company, Ch. Theoretical aspects of evolutionary multiobjective optimization, pp. 101–139.
- [6] Coelho, V. N., Souza, M. J. F., Coelho, I. M., Guimarães, F. G., Lust, T., Cruz, R. C., 2012. Multi-objective approaches for the open-pit mining operational planning problem. *Electronic Notes in Discrete Mathematics* 39, 233–240.
- [7] Deb, K., 2001. Multi-objective optimization using evolutionary algorithms. Wiley, New-York.
- [8] Drozdík, M., Akimoto, Y., Aguirre, H., Tanaka, K., 2015. Computational cost reduction of nondominated sorting using the M-front. *IEEE Transactions on Evolutionary Computation* 19 (5), 659–678.
- [9] Dubois-Lacoste, J., López-Ibáñez, M., Stützle, T., 2011. A hybrid TP+PLS algorithm for bi-objective flow-shop scheduling problems. *Computers & Operations Research* 38 (8), 1219–1236.
- [10] Fieldsend, J. E., Everson, R. M., Singh, S., 2003. Using unconstrained elite archives for multiobjective optimization. *IEEE Transactions on Evolutionary Computation* 7 (3), 305–323.
- [11] Fortin, F.-A., Grenier, S., Parizéau, M., 2013. Generalizing the improved run-time complexity algorithm for non-dominated sorting. In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation. GECCO ’13*. ACM, New York, NY, USA, pp. 615–622.
- [12] Habenicht, W., 1983. Essays and Surveys on Multiple Criteria Decision Making: *Proceedings of the Fifth International Conference on Multiple Criteria Decision Making*, Mons, Belgium, August 9–13, 1982. Springer Berlin Heidelberg, Ch. Quad Trees, a Datastructure for Discrete Vector Optimization Problems, pp. 136–145.

- [13] Inja, M., Kooijman, C., de Waard, M., Roijers, D., Whiteson, S., September 2014. Queued Pareto local search for multi-objective optimization. In: PPSN 2014: Proceedings of the Thirteenth International Conference on Parallel Problem Solving from Nature. pp. 589–599.
- [14] Jensen, M., Oct 2003. Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. *IEEE Transactions on Evolutionary Computation* 7 (5), 503–515.
- [15] Ke, L., Zhang, Q., Battiti, R., Oct 2014. Hybridization of decomposition and local search for multiobjective optimization. *IEEE Transactions on Cybernetics* 44 (10), 1808–1820.
- [16] Kooijman, C., de Waard, M., Inja, M., Roijers, D., Whiteson, S., April 2015. Pareto local policy search for MOMDP planning. In: ESANN 2015: Proceedings of the 23rd European Symposium on Artificial Neural Networks, Special Session on Emerging Techniques and Applications in Multi-Objective Reinforcement Learning. pp. 53–58.
- [17] Lin, S., Kernighan, B., 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research* 21, 498–516.
- [18] Lust, T., Jaszkiwicz, A., 2010. Speed-up techniques for solving large-scale biobjective TSP. *Computers & Operations Research* 37, 521–533.
- [19] Lust, T., Teghem, J., 2010. Two-phase Pareto local search for the biobjective traveling salesman problem. *Journal of Heuristics* 16 (3), 475–510.
- [20] Lust, T., Teghem, J., 2012. The multiobjective multidimensional knapsack problem: a survey and a new approach. *International Transactions in Operational Research* 19 (4), 495–520.
- [21] Lust, T., Tuytens, D., 2014. Variable and large neighborhood search to solve the multiobjective set covering problem. *J. Heuristics* 20 (2), 165–188.
- [22] Mostaghim, S; Teich, J., 2004. Quad-trees: A data structure for storing Pareto sets in multiobjective evolutionary algorithms with elitism. In: Abraham, A; Jain, L. G. R. (Ed.), *Evolutionary Multiobjective Optimization*, Book Series: Advanced Information and Knowledge. Springer-Verlag London LTD, pp. 81–104.
- [23] Mostaghim, S., Teich, J., Tyagi, A., 2002. Comparison of data structures for storing Pareto-sets in MOEAs. In: *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*. Vol. 1. pp. 843–848.
- [24] O’Mahony, C., Wilson, N., 2013. Sorted-pareto dominance and qualitative notions of optimality. In: Gaag, L. C. (Ed.), *Symbolic and Quantitative Approaches to Reasoning with Uncertainty: 12th European Conference, ECSQARU 2013, Utrecht, The Netherlands, July 8-10, 2013. Proceedings*. Springer Berlin Heidelberg, pp. 449–460.

- [25] Paquete, L., Chiarandini, M., Stützle, T., 2004. Pareto local optimum sets in the biobjective traveling salesman problem: an experimental study. In: Gandibleux, X., Sevaux, M., Sörensen, K., T'kindt, V. (Eds.), *Metaheuristics for Multiobjective Optimisation*. Springer. Lecture Notes in Economics and Mathematical Systems Vol. 535, Berlin, pp. 177–199.
- [26] Perny, P., Spanjaard, O., Storme, L.-X., 2006. A decision-theoretic approach to robust optimization in multivalued graphs. *Annals OR* 147 (1), 317–341.
- [27] Schütze, O., 2003. A new data structure for the nondominance problem in multi-objective optimization. In: *Proceedings of the 2nd International Conference on Evolutionary Multi-criterion Optimization. EMO'03*. Springer-Verlag, Berlin, Heidelberg, pp. 509–518.
- [28] Schütze, O., Mostaghim, S., Dellnitz, M., Teich, J., 2003. Covering Pareto sets by multi-level evolutionary subdivision techniques. In: *Proceedings of the Second Int. Conf. on Evolutionary Multi-Criterion Optimization*. Springer, Berlin, pp. 118–132.
- [29] Sun, M., Steuer, R., 1996. Quad trees and linear list for identifying nondominated criterion vectors. *INFORM Journal on Computing* 8 (4), 367–375.
- [30] Teixeira, C., Covas, J., Stützle, T., Gaspar-Cunha, A., 2009. Application of Pareto local search and multi-objective ant colony algorithms to the optimization of co-rotating twin screw extruders. In: Viana, A., et al. (Eds.), *Proceedings of the EU/Meeting 2009: Debating the future: new areas of application and innovative approaches*. pp. 115–120.
- [31] Zitzler, E., Deb, K., Thiele, L., 2000. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation* 8 (2), 173–195.